

Advanced Robot Control

Real-Time Operating System

Wojciech Domski

Chair of Cybernetics and Robotics,
Wrocław University of Science and Technology



Wrocław University
of Science and Technology



- 1 Operating system
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 FreeRTOS
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers
 - Heap management



Operating System (1/2)

What is an operating system?



Operating System (2/2)

It is a specific software that runs in kernel mode (supervised mode) [3]. The operating systems provides a set of API for using resources and managing these hardware resources.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers
 - Task management



OS support features in Cortex-M3/M4 core (1/3)

Cortex-M3/M4 support following features to facilitate the implementation of an embedded OS [4]:

- **Shadowed stack pointer**
The Main Stack Pointer (MSP) is used for the OS Kernel and interrupt handlers. The Processor Stack Pointer (PSP) is used by application tasks.
- **SysTick timer** A simple timer included inside the processor. This enables an embedded OS to be used on the wide range of Cortex-M microcontrollers available.



OS support features in Cortex-M3/M4 core (2/3)

- **Supervisor Call (SVC) and Pendable Service Call (PendSV) exceptions**

These two exception types are essential for the operations of embedded OSs such as the implementation of context switching.

- **Unprivileged execution level**

This allows a basic security model that restricts the access rights of some application tasks. The privileged and unprivileged separation can also be used in conjunction with the Memory Protection Unit (MPU), thus further enhancing the robustness of embedded systems.



OS support features in Cortex-M3/M4 core (3/3)

- **Exclusive accesses**

The exclusive load and store instructions are useful for semaphore and mutual exclusive (MUTEX) operations in the OS.



Outline

- 1 **Operating system**
 - OS support features
 - **Real-Time Operating System**
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers
 - Task management



Real-Time Operating System

The RTOS is a special kind of an operating system which focuses on time. Two groups of RTOS can be specified:

- soft real-time operating system,
- hard real-time operating system.



Hard RTOS

Hard real-time operating system is a specific branch of RTOS which has to meet strict time constraints.

This kind of OS can be usually found in industrial process control, avionics, military or any similar application where time is the critical factor. In this type of OS the action has to occur in a specific moment of time or in a specific time range.

If this condition is violated it can lead to a crash of whole application. Thus a hard RTOS should provide a certain mechanism which ensure this strict requirement.



Soft RTOS

In contrast to Hard RTOS a Soft RTOS is an operating system where occasional violation of strict time regime is undesirable but will not lead to any permanent damage. Entertainment devices such as music/video players and smartphones are example of a Soft RTOS.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - **Shadowed stack pointer**
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers



Main Stack Pointer and Processor Stack Pointer

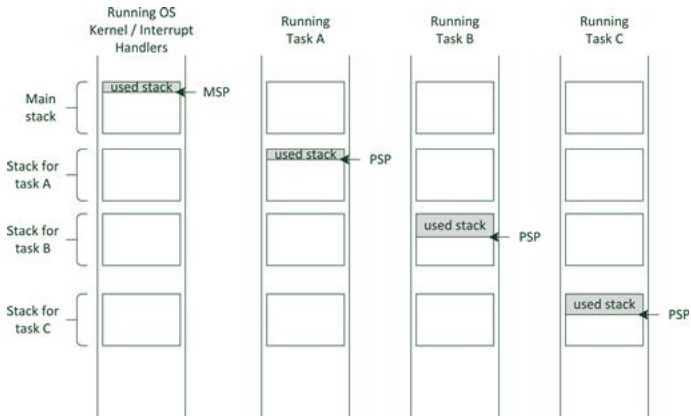


Figure: Stack pointer in a task [4]



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - **Supervised and unsupervised mode**
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers



Handler mode and Thread mode (1/1)

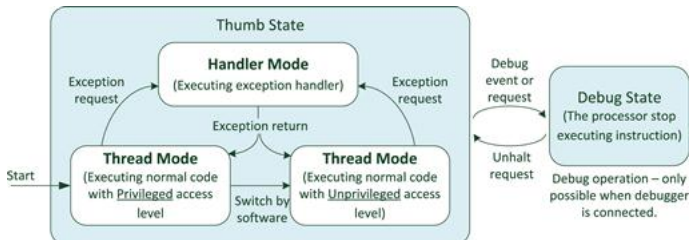


Figure: Operation states and modes [4]



Memory Protection Unit (1/2)

The MPU is a programmable device that can be used to define memory access permissions (e.g., privileged access only or full access) and memory attributes (e.g., bufferable, cacheable) for different memory regions [4].

The MPU on CortexM3 and Cortex-M4 processors can support up to eight programmable memory regions, each with their own programmable starting addresses, sizes, and settings. It also supports a background region feature (has the same memory access attributes as the default memory map, but is accessible from privileged software only).



Memory Protection Unit (2/2)

The MPU can be used to make an embedded system more robust , and in some cases it can make the system more secure by:

- Preventing application tasks from corrupting stack or data memory used by other tasks and the OS kernel
- Preventing unprivileged tasks from accessing certain peripherals that can be critical to the reliability or security of the system
- Defining SRAM or RAM space as non-executable (eXecute Never, XN) to prevent code injection attacks



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - **Context switching**
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers



Starting task with SVC

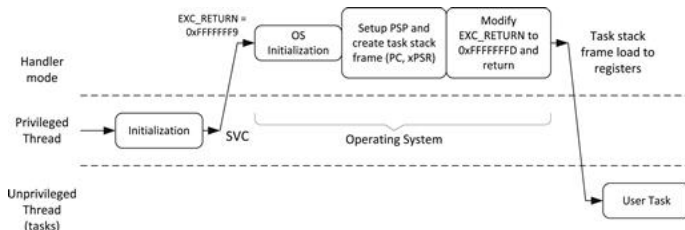


Figure: Initialization of a task [4]



Switching task with PendSV

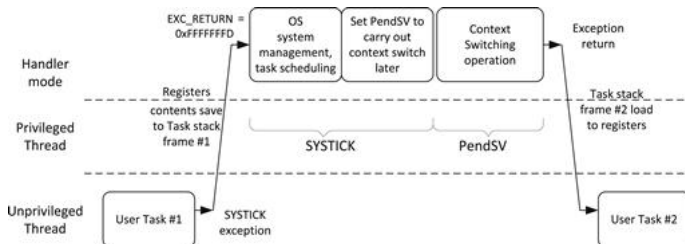


Figure: Context switching [4]



Switching task during interrupt (1/2)

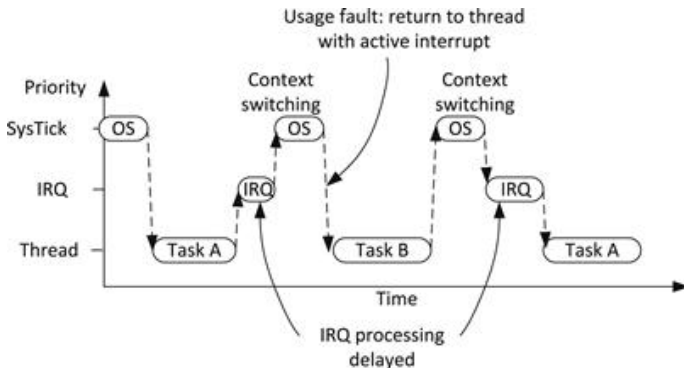


Figure: Improper task switching [4]



Switching task during interrupt (2/2)

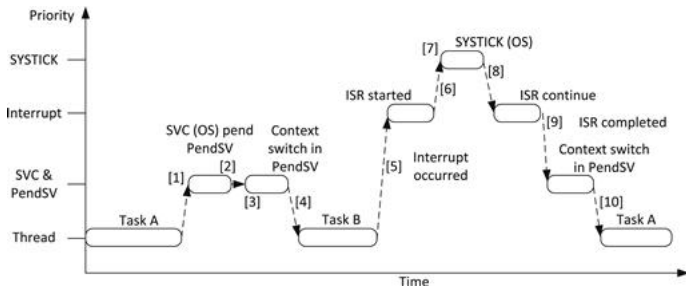


Figure: Proper task switching [4]



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - **Exclusive accesses**
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers



Hardware Mutex support (1/4)

A common way to limit the access to a resource is a semaphore. It can be represented as a counter which holds a number of resource. To obtain access to the resource following has to happen:

- 1 Read current number of the variable and if it is greater than zero go to the next point.
- 2 Decrement the value by one.
- 3 Write the new value to the variable.

The problem arises when there is a context switch between any point. The wrong value is calculated and the semaphore value is not reassembling the current situation.

Also, special case of a semaphore with only one resource available is a mutex, mutual exclusion type of a barrier.



Hardware Mutex support (2/4)

The Cortex-M3/M4 support a feature called exclusive access to facilitate the implementation of barriers.

The semaphore variables are read and written using exclusive load and exclusive store. If during the store operation it was found that the access cannot be guaranteed to be exclusive, the exclusive store fails and the write will not take place. The processor should then retry the exclusive access sequence.



Hardware Mutex support (3/4)

There is a dedicated unit called local monitor which watches over the task of the exclusive access.

There are three following instructions that allow the implementation of a semaphore:

- STREX – stores data,
- CLREX – switches local monitor to Open Access,
- LDREX – switches local monitor to Exclusive Access.



Hardware Mutex support (4/4)

The exclusive write (e.g., STREX) can fail if one of the following operations has taken place:

- A CLREX instruction has been executed, switching the local monitor to the Open Access state.
- A context switch has occurred (e.g., an interrupt).
- There wasn't a LDREX executed beforehand.
- An external hardware returns an exclusive fail status to the processor via a sideband signal on the bus interface.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - **Introduction**
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers
 - Task management



Introduction

FreeRTOS is a real-time kernel on top of which embedded applications can be built to meet their hard real-time requirements [1].

It allows applications to be organized as a collection of independent threads of execution. On a processor that has only one core, only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer.



Features (1/6)

- **Abstracting away timing information**

The kernel is responsible for execution timing and provides a time-related API to the application. This allows the structure of the application code to be simpler, and the overall code size to be smaller.

- **Maintainability/Extensibility**

Abstracting away timing details results in fewer interdependencies between modules, and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.



Features (2/6)

- **Modularity**

Tasks are independent modules, each of which should have a well-defined purpose.

- **Team development**

Tasks should also have well-defined interfaces, allowing easier development by teams.

- **Easier testing**

Tasks should also have well-defined interfaces, allowing easier development by teams.

- **Code reuse**

Greater modularity and fewer interdependencies results in code that can be reused with less effort.



Features (3/6)

- **Improved efficiency**

Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.

Counter to the efficiency saving is the need to process the RTOS tick interrupt, and to switch execution from one task to another. However, applications that don't make use of an RTOS normally include some form of tick interrupt anyway.



Features (4/6)

- **Idle time**

The Idle task is created automatically when the scheduler is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.



Features (5/6)

- **Power Management**

The efficiency gains that are obtained by using an RTOS allow the processor to spend more time in a low power mode.

Power consumption can be decreased significantly by placing the processor into a low power state each time the Idle task runs. FreeRTOS also has a special tick-less mode. Using the tick-less mode allows the processor to enter a lower power mode than would otherwise be possible, and remain in the low power mode for longer.



Features (6/6)

- **Flexible interrupt handling**

Interrupt handlers can be kept very short by deferring processing to either a task created by the application writer, or the FreeRTOS daemon task.

- **Mixed processing requirements**

Simple design patterns can achieve a mix of periodic, continuous and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities



ISR dedicated API

ISR specific API

It is worthy of note that not all API can be called in ISR routines. Most of API available in FreeRTOS has two implementations. The first group is dedicated for a normal in-task usage while the second group is used only in ISRs. The latter representatives have usually a ISR suffix in the name of a function.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - **Tasks**
 - Mutexes
 - Queues
 - Event groups
 - Software timers
 - Task management



Task states

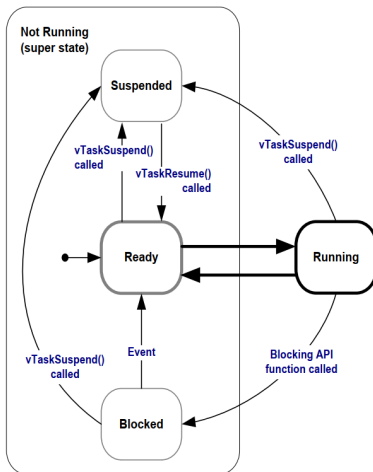


Figure: Task switching [1]



Task creation

Task is created using following function

```

1 BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
2     const char * const pcName,
3     uint16_t usStackDepth,
4     void *pvParameters,
5     UBaseType_t uxPriority,
6     TaskHandle_t *pxCreatedTask );

```

The task definition has to be following

```

1 void Task( void *pvParameters ) {
2     for (;;) {
3     }
4 }

```

After all task are created (typical situation) the scheduler can be started with



vTaskStartScheduler();
 of Science and Technology



Delaying task execution

Task can be delayed with

```
1 void vTaskDelay( TickType_t xTicksToDelay );
```

but it does not guarantee that the task will be periodic.
Implementation of a periodic task can be done with

```
1 void vTaskDelayUntil( TickType_t * pxPreviousWakeTime ,  
2                       TickType_t xTimeIncrement );
```

Above function will adjust necessary delay period. It is done thanks to storing last wake-up time

```
1 TickType_t xLastWakeTime ;  
2 xLastWakeTime = xTaskGetTickCount() ;
```

Using busy waits is not preferred. Depending on priorities one task can starve another.



Outline

- 1 Operating system
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 FreeRTOS
 - Introduction
 - Tasks
 - **Mutexes**
 - Queues
 - Event groups
 - Software timers
 - Semaphore management



Mutexes (1/2)

A mutex can be created using [2]

```
1 SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

To take (lock) mutex following function has to be called

```
1 BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,  
    TickType_t xTicksToWait );
```

while it can be released (unlocked) with

```
1 BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```



Mutexes (2/2)

During usage of mutexes a situation called *Deadlock* can arise. When a task A is waiting for resources which already has been taken by an another task B, and in turn task B is waiting for resources claimed by task A.



Critical section

A critical section is a region of code surrounded with

```
1 taskENTER_CRITICAL () ;  
2 // ...  
3 taskEXIT_CRITICAL () ;
```

This allows to disable interrupts completely or up to a defined interrupt priority. To facilitate this mechanism **BASEPRI** masking is used.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - **Queues**
 - Event groups
 - Software timers
 - Task management



Queues (1/4)

Queues are normally used as **First In First Out (FIFO) buffers**, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.

Queue can be created with

```
1 QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,  
2 UBaseType_t uxItemSize );
```

Message can be send to queue with

```
1 BaseType_t xQueueSendToFront( QueueHandle_t xQueue,  
2 const void * pvItemToQueue,  
3 TickType_t xTicksToWait );
```

or (typically)

```
1 BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
2 const void * pvItemToQueue,  
3 TickType_t xTicksToWait );
```



Queues (2/4)

A message from queue can be read using

```
1 BaseType_t xQueueReceive( QueueHandle_t xQueue,  
2 void * const pvBuffer ,  
3 TickType_t xTicksToWait );
```

Current number of items inside a queue can be checked with

```
1 UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```



Queues (3/4)

Queue can be used as a **message box** where size of a queue is equal to 1.

Current message can be replaced with

```
1 BaseType_t xQueueOverwrite( QueueHandle_t xQueue,  
2 const void * pvItemToQueue );
```

To read a message from a queue function `xQueueReceive()` can be used or if it is required to retain the message in the queue below function can be used instead

```
1 BaseType_t xQueuePeek( QueueHandle_t xQueue,  
2 void * const pvBuffer,  
3 TickType_t xTicksToWait );
```



Queues (4/4)

Sending large portion of data through queue

It is recommended to send a pointer to an area containing the data rather than sending a large portion of data via a queue. For this purpose buffer polls can be used.



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - **Event groups**
 - Software timers
 - Heap management



Event groups (1/2)

Event group is a mechanism available in FreeRTOS which allows to suspend a task in a blocking state as long as a certain condition is not met.

An event group can be represented as a variable where each bit is called a flag. A flag can be set to logical '1'. A task can wait for a certain configuration of bits in this variable to be set or at least one the flags.



Event groups (2/2)

```
1 EventGroupHandle_t xEventGroupCreate( void );
```

Above creates an event group.

Bits in an event group can be set using

```
1 EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup ,
2 const EventBits_t uxBitsToSet );
```

A task is suspended when function

```
1 EventBits_t xEventGroupWaitBits( const EventGroupHandle_t
   xEventGroup ,
2 const EventBits_t uxBitsToWaitFor ,
3 const BaseType_t xClearOnExit ,
4 const BaseType_t xWaitForAllBits ,
5 TickType_t xTicksToWait );
```

is called. The `uxBitsToWaitFor` is a mask through which event can be identified. Function can wait for all flags (bits) to be set or not (`xWaitForAllBits`).



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - **Software timers**



Software timers (1/4)

Software timers can run a function (callback function) at a specific time or periodically.

The software timers run independently from the hardware and are governed by the FreeRTOS kernel.

The time grain is equal to the FreeRTOS time period.



Software timers (2/4)

Timer can be created with

```
1 TimerHandle_t xTimerCreate( const char * const pcTimerName,  
2 TickType_t xTimerPeriodInTicks,  
3 UBaseType_t uxAutoReload,  
4 void * pvTimerID,  
5 TimerCallbackFunction_t pxCallbackFunction );
```

while it can be started with

```
1 BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t  
xTicksToWait );
```

and stopped with

```
1 BaseType_t xTimerStop( TimerHandle_t xTimer, TickType_t  
xTicksToWait );
```



Software timers (3/4)

Each software timer has an ID (`pvTimerID`) which can be used for any purpose. It is set during the creation of a software timer and can be changed directly with

```
1 void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

The timer ID can be retrieved with

```
1 void *pvTimerGetTimerID( TimerHandle_t xTimer );
```



Software timers (4/4)

Timer's period can be changed with

- 1 BaseType_t xTimerChangePeriod(TimerHandle_t xTimer ,
- 2 TickType_t xNewTimerPeriodInTicks ,
- 3 TickType_t xTicksToWait);

Timer can be reset with

- 1 BaseType_t xTimerReset(TimerHandle_t xTimer , TickType_t
xTicksToWait);



State machine for one shot sw timer (1/1)

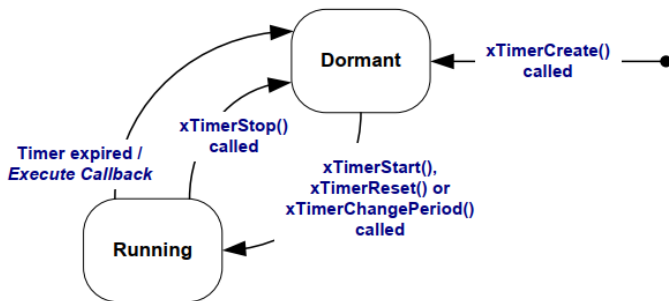


Figure: One-shot software timer states and transitions [1]



State machine for auto-reload sw timer (1/1)

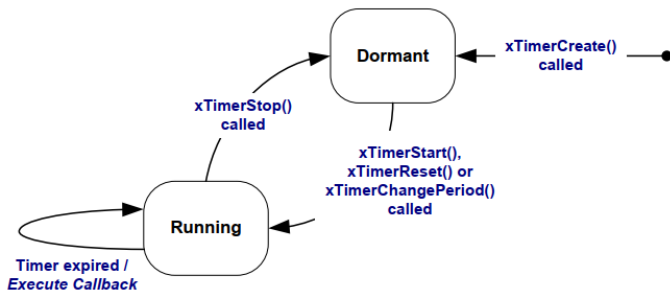


Figure: Auto-reload software timer states and transitions [1]



Outline

- 1 **Operating system**
 - OS support features
 - Real-Time Operating System
 - Shadowed stack pointer
 - Supervised and unsupervised mode
 - Context switching
 - Exclusive accesses
- 2 **FreeRTOS**
 - Introduction
 - Tasks
 - Mutexes
 - Queues
 - Event groups
 - Software timers



Heap implementations

In FreeRTOS a number of heap implementations is available:

- **Heap_1** – does not implement freeing memory and the memory is of **static size** determined by the `configTOTAL_HEAP_SIZE`.
- **Heap_2** – similar to **Heap_1** but allows memory to be deallocated, uses a best fit algorithm to allocate memory.
- **Heap_3** – standard implementation of *malloc()* and *free()*. Memory is allocated dynamically. The size of the heap is defined through linker configuration.
- **Heap_4** – similar to **Heap_1** and **Heap_2** but a first fit algorithm and the space is divided into smaller chunks.
- **Heap_5** – similar to **Heap_4** in terms of an algorithm used for allocation and deallocation of memory. However, allows memory to be allocated from multiple and separate regions.



Literature (1/2)



R. Barry.

Mastering the FreeRTOS™ Real Time Kernel.
Real Time Engineers Ltd., 2016.



Amazon Web Services.

The FreeRTOS™ Reference Manual, API Functions and Configuration Options.
Amazon.com Inc., 2017.



A. S. Tanenbaum and H. Bos.

Moder Operating systems.
Pearson, 2014.



J. Yiu.

The Definitive Guide to ARM® Cortex® -M3 and Cortex® -M4 Processors.
Newnes, 2013.



Literature (2/2)

